**Контейнеры**

**pair**

В программировании часто возникает необходимость обрабатывать два объекта, например, простую пару чисел, как один. Такая задача стала особенно актуальной, когда в синтаксис программирования были внедрены [ассоциативные массивы](https://proginfo.ru/map/) с их парой: название поля –> значение поля. Причём типы обоих значений в паре могут быть совершенно разными – в этом и состоит особая ценность рассматриваемой [структуры](https://proginfo.ru/struct/) и на это необходимо обратить особое внимание.

Откликом на такую задачу одновременной обработки пары значений стало появление структуры **pair**, описание которой приводится ниже.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | // структура шаблона хранит пары объектов типа A1 и A2;  // тип fsttype совпадает с типом параметра шаблона A1, а тип sectype – A2;  // таким образом, на практике A1 может соответствовать имени поля,  // а A2 – его значению    template <class A1, class A2>    // все члены типа pair являются открытыми - объявлены как struct, а не как class    struct pair  { typedef fsttype A1;    typedef sectype A2;    A1 first;                            // 1-ый элемент пары fst имеет тип A1    A2 second;                           // 2-ой элемент пары sec – A2      // конструктор по умолчанию - инициализирует 1-ый элемент пары как тип по умолчанию A1, а 2-ой элемент – как A2      pair ();      // конструктор инициализирует 1-ый элемент пары как v1, а 2-ой элемент - как v2      pair (const A1 &v1, const A2 &v2);      // конструктор (шаблон) инициализирует 1-ый элемент пары как elem.fst,    // а 2-ой - как elem.sec      template <class b1, class b2>    pair (const pair <b1, b2> &elem );      // конструктор инициализирует 1-ый элемент пары как v1,    // а 2-ой элемент - как v2      template <class b1, class b2>    pair (const pair <b1 v1, b2 v2> &elem );  } |

Реализация сделанных объявлений выглядит следующим образом.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | #include <utility>  #include <map>  #include <iomanip>  #include <iostream>  #include <conio.h>  #include <windows.h>    int main ()  {    SetConsoleCP(1251);    SetConsoleOutputCP(1251);    setlocale(LC\_ALL,"russian");    using namespace std; // используемое пространство имён      pair <int, double> p1 (10, 0.011); // через конструктор инициализируем пару  // p1, в которой 1-ый элемент имеет целый  // тип, а 2-ой – действительный двойной  // точности      pair <int, double> p2; // здесь просто инициализируем пару p2 без  // присвоения его двум полям конкретных  // значений    p2 = make\_pair (10, 0.222); // какая-то ф-ция, обеспечивающая  // присвоение значений полям пары p2      pair <int, double> p3 (p1); // инициализация пары p3, как копии уже  // созданной p1      cout.precision (3); // назначаем точность вывода  // действительных чисел      // проверяем результаты работы выводом на устройство вывода по умолчанию      cout << "Пара p1 состоит из элементов : ( " << p1.first << ", " << p1.second << " )"            << endl;    cout << "- p2 : ( " << p2.first << ", " << p2.second << " )" << endl;    cout << "- p3 : ( " << p3.first << ", " << p3.second << " )" << endl;      // пример более сложного применения pair, когда структура представляет собой    // массив пар, оба элемента в которой целые числа (на примере карты памяти)      map <int, int> mydb; // это наша база данных, как карта памяти    map <int, int> :: iterator mydbindex;      typedef pair <int, int> mypair; // определяет тип пары mypair, в которой  // каждые 2 элемента – целые числа      mydb.insert ( mypair (1, 10) ); // одновременно, через конструктор  // создаётся пара, её элементам  // присваиваются значения 1 и 10 и  // она вставляется в базу  // Внимание! Предполагается, что функция  // вставки insert() реализована в базе map    mydb.insert ( mypair (2, 20) );    mydb.insert ( mypair (3, 30) ); // после данной вставки в базе уже 3 пары  // элементов      cout << "База mydb состоит из следующих элементов : ";      // просматриваем всю базу через её индекс,    // получая 3 пары в формате «(число, число)»    // Внимание! Предполагается, что функции begin() и end() в базе реализованы      for ( mydbindex = mydb.begin(); mydbindex != mydb.end(); mydbindex++ )    cout << "(" << mydbindex -> first << ", " << mydbindex -> second << ")";      // Внимание! Обратите внимание на следующее объявление!    // Создаются две пары res1 и res2, как результат вставки нового элемента в базу mydb    // 1-ое поле каждой пары индексом на вставленную позицию в пару, а    // 2-ое отвечает за результат вставки – успешна/неуспешна – тип логический      pair <map <int, int> :: iterator, bool> res1, res2;      res1 = mydb.insert ( mypair (4, 40) ); // помимо собственно вставки в базу  // пары 4 и 40 insert() выдаёт true или false  // в случае успеха или неуспеха операции    res2 = mydb.insert ( mypair (1, 10) );      if (res1.second == true)        { cout << "Пара (4, 40) успешно вставлена в базу" << endl;    }    else { cout << "Пара с ключевым первым элементом " << " ( (res1.fst) -> fst ) = "     << ( res1.first ) -> first  << " уже есть в базе, вставка невозможна" << endl;        }      if (res2.second == true)        { cout << "Пара (1, 10) успешно вставлена в базу" << endl;        }    else { cout << "Пара с ключевым первым элементом " << " ( (res2.fst) -> fst ) = "     << ( res2.first ) -> first  << " уже есть в базе, вставка невозможна" << endl;    }    getch();  } |

Заметим, пара (4, 40) будет вставлена успешно, а (1, 10) – нет, пара (или запись, если переходить к терминологии баз данных) с ключом «1» уже в базе есть. Такое поведение один к одному соответствует поведению любой базы данных, когда она принимает данные с ключевыми полями. Собственно в этом и заключается особая ценность структуры pair и принципов работы с ней.

**vector**

Вектор в языке C++ - это абстрактная модель с рядом определенных [функций](https://proginfo.ru/functions/), позволяющих совершать различные действия, в том числе имитировать работу [динамического массива](https://proginfo.ru/dynamic_array/).

Что бы использовать вектора в программе, необходимо [подключить](https://proginfo.ru/add_header/) заголовочный файл <vector>.  Его объявление выглядит следующим образом

|  |  |
| --- | --- |
| 1 | vector<int> myV (15); |

В этой строке <int> означает [тип](https://proginfo.ru/data-types/) элементов в векторе. Все элементы вектора должны принадлежать одному типу. Число 15 означает, что в векторе задается память на 15 элементов, при этом они инициализируются. Возможна и другая запись

|  |  |
| --- | --- |
| 1  2 | vector<int> myV;  myV.reserve(15); |

В ней выделяется память на 15 элементов уже без инициализации.

Работать с вектором можно как с [массивом](https://proginfo.ru/arrays/), так и с объектом. Для вектора как объекта функции таковы:

* myV.swap(myV2) - меняет содержимое векторов местами,
* myV.begin() - указатель на начало вектора,
* myV.end() - указатель на конец вектора,
* myV.rbegin() - реверсивный указатель на конец вектора,
* myV.rend() - реверсивный указатель на начало вектора,
* myV.clear() - очистка вектора.

Вектор также  примечателен тем, что при добавлении в него элементов больше, чем изначально задано, он изменяет свой размер и резервирует дополнительную память. Функций для такой работы с памятью вектора немного:

* myV.size() - размер вектора,
* myV.max\_size() - максимальный размер вектора,
* myV.reserve(n) - установить минимальный размер выделенной памяти на вектор,
* myV.resize(n) - установить размер n для вектора,
* myV.capacity() - количество свободной памяти, выделенной под вектор.

В целом, он служит неплохой альтернативой массивам, позволяя более удобно и быстро работать.

Ну и основные функции, для работы с отдельными элементами вектора:

* myV.push\_back(e) - добавить в конец элемент,
* myV.pop\_back() - удалить последний элемент,
* myV.front() - первый элемент вектора,
* myV.back() - последний элемент вектора,
* myV.insert(i, e) - вставка элемента в позицию i,
* myV.erase(myV2) - удаляет последовательность элементов,
* myV[i] или myV.at(i) - доступ к элементу i вектора.

Примеры использования этих функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #include <vector> // подключаем vector  #include <string> // подключаем строки string  #include <iostream> // подключаем cout    using namespace std; // чтобы не писать std::    int main()  {    vector <string> V; // инициализация vector      V.push\_back("January");    V.push\_back("March");    V.push\_back("April");    V.insert(V.begin()+1,"February");    V.push\_back("May");    V.pop\_back();    cout<<V.size()<<'\n'; // будет выведено 4    cout<<V.max\_size()<<'\n';    cout<<\*(V.begin())<<'\n'; // January    cout<<\*(V.end())<<'\n'; // NULL    cout<<\*(V.rbegin())<<'\n'; // April    cout<<V.back()<<'\n'; // April    cout<<V.front()<<'\n'; // January    V.erase(V.begin()+1);    cout<<V[1]<<'\n'; // March    V.clear();    cout<<V.size()<<'\n'; // 0    cin.get(); // ожидаем нажатие пользователем клавиши      return 0; // выдаём 0 - правило хорошего тона  } |

**Динамический массив** - это [массив](https://proginfo.ru/arrays/), в котором количество элементов, и соответственно выделенный на него объем памяти, может меняться как при инициализации, так и при работе с ним. Строго говоря, такой массив не реализован в языке C++. Есть некоторая его замена, обладающая рядом функций динамического массива.

Объявляют  массив таким образом:

|  |  |
| --- | --- |
| 1  2  3 | int a;  a = 7;  int \*mas = new int[a]; |

Благодаря такой конструкции можно инициализировать массив любого размера, но изменять размер выделенной на него памяти в ходе работы невозможно. Можно только полностью освободить занимаемое массивом место. Это делается с помощью функции **delete**. Для нашего примера это выглядит так

|  |  |
| --- | --- |
| 1 | delete [] mass; |

Скобки перед оператором обязательны, они указывают на то, что удаляется именно массив, а не переменная.

Динамическим можно сделать не только одномерный массив, но и [двумерный](https://proginfo.ru/matrix/) и выше. Так, например, двумерный динамический массив:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int a = 2;  int b = 3;    float \*\*ptrarray = new float\* [a]; // инициализация  for (int count = 0; count < a; count++)     ptrarray[count] = new float [b];    for (int count = 0; count < a; count++) // освобождение памяти     delete [] ptrarray[count];  delete [] ptrarray; |

Обратиться к элементу динамического массива можно также как и к элементу обычного массива:

|  |  |
| --- | --- |
| 1  2 | a[i] = 5;  ptrarray[1][2] = 4; |

Так же некоторой альтернативой полноценному динамическому массиву может являться объект [**vector**](https://proginfo.ru/vector/). Он обладает функционалом обычных массивов C++, но при этом может менять размер при добавлении в него новых элементов. Например,

|  |  |
| --- | --- |
| 1  2 | vector<int> mV;  mV.push\_back(5); |

добавляет в конец пустого массива элемент 5, тем самым увеличивая его размер с 0 до 1.

[VK](https://proginfo.ru/#vk)[Отправить](https://www.addtoany.com/share#url=https%3A%2F%2Fproginfo.ru%2Fdynamic_array%2F&title=%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9%20%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2)

**list**

В С++ контейнер list задаёт стандартные двунаправленные списки. В эти списки можно быстро вставлять, а также удалять элементы. Однако операция обращения к элементу по номеру долгая.

Использование list требует [подключения](https://proginfo.ru/add_header/)

|  |  |
| --- | --- |
| 1 | #include <list> |

Объявляется список так:

|  |  |
| --- | --- |
| 1 | list<int> MyL; |

При работе со списком можно воспользоваться указателями:

* myL.begin() - указатель на начало списка,
* myL.end() - указатель на конец списка,
* myL.rbegin() - реверсивный указатель на конец списка,
* myL.rend() - реверсивный указатель на начало списка.

Полезна также функция проверки списка на пустоту:

* myL.empty().

Основные функции, для работы с отдельными элементами списка:

* myL.push\_back(e) - добавить в конец элемент,
* myL.pop\_back() - удалить последний элемент,
* myL.front() - первый элемент списка,
* myL.back() - последний элемент списка,
* myL.insert(i, e) - вставка элемента в позицию i,
* myL.erase(first,last) - удаляет последовательность элементов.

При работе со списками, чтобы удобно было по списку ходить, можно пользоваться итераторами, например,

|  |  |
| --- | --- |
| 1 | list<int>::iterator i; |

Наконец, рассмотрим пример работы со списком:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <list> // подключаем list  #include <string> // подключаем строки string  #include <iostream> // подключаем cout    using namespace std; // чтобы не писать std::    int main()  {    list<string> L; // инициализация list    list<string>::iterator i;      L.push\_back("January"); // список January    L.push\_back("March"); // список January March    L.push\_back("April"); // список January March April      i = L.begin();    i++;    L.insert(i, "February"); // список January February March April      L.push\_back("June"); // список January February March April June      i = L.end();    i--;    L.insert(i, "May"); // список January February March April May June      i = L.end();    i--;    i--;    i--;    L.erase(i); // список January February March May June      L.pop\_back(); // список January February March May      for(i = L.begin(); i != L.end(); i++)      cout << (\*i) << " ";      system("pause");  } |

**set**

В языке C++ контейнер **set** позволяет работать с различными множествами. Под множеством в данном случае понимают некоторое количество отсортированных элементов. Использование  в файле set требует [подключения](https://proginfo.ru/add_header/)

|  |  |
| --- | --- |
| 1 | #include <set> |

В данном контейнере сортировка происходит автоматически, при добавлении элемента. Причем повторяющиеся элементы могут быть, как удалены, в случае работы с множеством, так и записаны, в случае с мультимножеством, позволяющим это. Рассмотрим конкретный пример

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | set<int> S;  multiset<int> mS;    mS.insert(1);  mS.insert(2);  mS.insert(2);  S.insert(1);  S.insert(2);  S.insert(2); |

После добавлений  мы получим 2 множества S = (1, 2)  и mS = (1, 2, 2). Вместо insert можно использовать функцию emplace, которая перед добавлением проверяет, есть ли уже такой элемент, и если нет, то добавляет его.

Set является контейнером, так что имеет все стандартные для контейнера функции:

- Работа с элементами set:

* S.swap(S2) - меняет содержимое контейнеров местами,
* S.insert(a) - вставка элемента a,
* S.erase(S2) - удаляет последовательность элементов,
* S.clear() - очистка контейнера set,
* S.count() - количество элементов в контейнере,
* S.find(a) - найти элемент a в контейнере,
* S.lower\_bound(a) - первый элемент, не меньший чем a,
* S.upper\_bound(a) - первый элемент, больший чем a,
* S.equal\_range(a) - пара элементов, первый - нижняя граница элементов с, такими же значениями, что и a, второй - верхняя граница элементов с такими же значениями, что и a.

- Работа с памятью:

* S.size() - размер контейнера,
* S.max\_size() - максимальный размер контейнера.

- Работа с контейнером:

* S.begin() - указатель на начало контейнера,
* S.end() - указатель на конец контейнера,
* S.rbegin() - реверсивный указатель на конец контейнера,
* S.rend() - реверсивный указатель на начало контейнера.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <set> // подключаем set  #include <string> // подключаем строки string  #include <iostream> // подключаем cout    using namespace std; // чтобы не писать std::    int main()  {  set <string> S; // инициализация set    S.insert("January");  S.insert("February");  S.insert("March");  S.insert("April");  cout<<S.size()<<'\n'; // будет выведено 4  cout<<S.max\_size()<<'\n';  cout<<\*(S.begin())<<'\n'; // April  cout<<\*(S.rbegin())<<'\n'; // March  cout<<S.count("January")<<'\n'; // 1  cout<<\*S.lower\_bound("January")<<'\n'; // January  cout<<\*S.upper\_bound("January")<<'\n'; // March  cout<<\*(S.equal\_range("January")).first<<'\n'; // January  cout<<\*(S.equal\_range("January")).second<<'\n'; // March  cout<<\*S.find("January")<<'\n'; // January  S.erase("January");  cout<<S.count("January")<<'\n'; // 0  S.clear();  cout<<S.size()<<'\n'; // 0  cin.get(); // ожидаем нажатие пользователем клавиши    return 0; // выдаём 0 - правило хорошего тона  } |

[VK](https://proginfo.ru/#vk)[Отправить](https://www.addtoany.com/share#url=https%3A%2F%2Fproginfo.ru%2Fset%2F&title=set)

**map**

Контейнер **map** в языке C++ позволяет работать сразу с 2 значениями. Каждый из элементов пары может иметь свой [тип](https://proginfo.ru/data-types/). Одним из основных способов использования map является создание ассоциативного массива. Это означает, что один из элементов пары становится ключом, по которому можно извлечь второй элемент, хранящий какую-либо информацию. Приведем пример

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #include <map>    map <string,int> Mv = {{ "Winter", 11 },                         { "Spring", 4 },                         { "Summer", 6 },                         { "Autumn", 10 }}; |

В примере мы явно задали [массив](https://proginfo.ru/arrays/) в данном контейнере, добавив в него 4 пары элементов. Все пары соответствуют изначально заданным типам <string,int>. В дальнейшем работа с Mv идет, как и с любым другим контейнером, посредством стандартных функций. Например,

|  |  |
| --- | --- |
| 1  2  3  4  5 | Mv.insert ( pair< string,int >(c,i) );  for (auto it = Mv.begin(); it != Mv.end(); ++it)  {     cout << (\*it).first << " : " << (\*it).second << endl;  } |

Первой строкой мы добавляем еще одну пару элементов. Обязательно, чтобы пара имела такие же типы, что и Mv. В строчках 2-5 с помощью цикла выводятся на экран  все пары элементов в контейнере, каждая пара на новой строке и разделена “:”. Первый и второй элементы пары вызываются  (\*it).first  и (\*it).second соответственно. Mv.begin() и Mv.end() указывают на начало и конец массива элементов контейнера. Map автоматически сортирует значения в массиве, и не позволяет добавлять идентичные пары. Это решается путем использования multimap:

|  |  |
| --- | --- |
| 1 | multimap <string,int> Mv; |

Функционал контейнера map:

- Работа с памятью:

* Mv.size() - размер контейнера,
* Mv.max\_size() - максимальный размер контейнера.

- Работа с контейнером:

* Mv.begin() - указатель на начало контейнера,
* Mv.end() - указатель на конец контейнера,
* Mv.rbegin() - реверсивный указатель на конец контейнера,
* Mv.rend() - реверсивный указатель на начало контейнера.

- Работа с элементами map:

* Mv.swap(Mv2) – меняет содержимое контейнеров местами,
* Mv.insert(e) - вставка элемента,
* Mv.erase(Mv2) - удаляет последовательность элементов,
* Mv[i] или Mv.at(i)  доступ к элементу i контейнера,
* Mv.clear() - очистка контейнера map,
* Mv.count() - количество элементов в контейнере,
* Mv.find(a) - найти элемент в контейнере,
* Mv.lower\_bound(a) - первый элемент, не меньший чем a,
* Mv.upper\_bound(a) - первый элемент, больший чем a,
* Mv.equal\_range(a) - пара элементов, первый - нижняя граница элементов с, такими же значениями, что и a, второй - верхняя граница элементов с такими же значениями, что и a,
* Mv.key\_comp() - сортировка элементов в контейнере по ключу,
* Mv.value\_comp() - сортировка элементов в контейнере по значению.

Примеры использования этого функционала:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <map> // подключаем map  #include <string> // подключаем строки string  #include <iostream> // подключаем cout    using namespace std; // чтобы не писать std::    int main()  {    map <string,int> Mv; // инициализация map      Mv.insert(pair<string,int>("January",31));    Mv.insert(pair<string,int>("February",28));    Mv.insert(pair<string,int>("March",31));    Mv.insert(pair<string,int>("April",30));    cout<<Mv.size()<<'\n'; // будет выведено 4    cout<<Mv.max\_size()<<'\n';    cout<<(\*(Mv.begin())).first<<'\n'; // April    cout<<(\*(Mv.end())).first<<'\n'; // NULL    cout<<(\*(Mv.rbegin())).first<<'\n'; // March    cout<<(\*(Mv.rend())).first<<'\n'; // January    cout<<Mv["January"]<<'\n'; // 31    cout<<Mv.count("January")<<'\n'; // 1    cout<<(\*Mv.lower\_bound("January")).first<<'\n'; // January    cout<<(\*Mv.upper\_bound("January")).first<<'\n'; // March    cout<<((Mv.equal\_range("January")).first)->first<<'\n'; // January    cout<<((Mv.equal\_range("January")).second)->first<<'\n'; // March    cout<<(\*Mv.find("January")).first<<'\n'; // January    cout<<Mv.key\_comp()("January","April")<<'\n'; // 0 (т.е. >)    cout<<Mv.value\_comp()(\*Mv.rbegin(),\*Mv.rend())<<'\n'; // 0 (т.е. >)    Mv.erase("January");    cout<<Mv.count("January")<<'\n'; // 0    Mv.clear();    cout<<Mv.size()<<'\n'; // 0    cin.get(); // ожидаем нажатие пользователем клавиши      return 0; // выдаём 0 - правило хорошего тона  } |

**Контейнеры: map, set, multiset**

std::map (ассоциативный массив)

Ассоциативный массив – структура данных, которая позволяет нам выполнять различные операции над парами <ключ, значение>. В С++ ассоциативные массивы представлены в стандартной библиотеке классом std::map.

Асимптотическая сложность

* Вставка элемента, обновление значения ключа – O(logN)O(log⁡N).
* Удаление по ключу – O(logN)O(log⁡N).
* Поиск по ключу – O(logN)O(log⁡N).
* Поиск по значению – O(N)O(N) (необходимо итерироваться, пока не будет найдено необходимое значение).
* Подсчёт различных ключей на текущий момент – O(1)O(1).

Стоит помнить, что асимптотическая сложность – по сути теоретический параметр, а на практике разница между временем работы разных операций одинаковой сложности может быть очень ощутима. В случае map, операции сложностью O(logN)O(log⁡N) работают гораздо медленнее, чем, например, бинарный поиск, и при решении задач об этом необходимо помнить.

Использование

Решим простую задачу, используя map:

Дана строка ss из nn слов. Необходимо вывести наиболее часто встречающееся слово и количество его вхождений в стрку.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **#include <bits/stdc++.h>**  **using** **namespace** std;  **int** **main**() {  *// Объявление: map<t1, t2>.*  *// t1 - тип ключа*  *// t2 - тип значения*  map**<**string, **int>** cnt;  **int** n;  string s;  cin **>>** n;  **for** (**int** i **=** 0; i **<** n; i **++**) {  cin **>>** s;  *// Подсчет количества вхождений каждого слова.*  cnt[s]**++**;  }  **int** entries **=** 0;  string word **=** "";  *// Итерирование по map:*  *// обход производится в порядке возрастания ключей.*  **for** (pair**<**string, **int>** i**:** cnt) {  **if** (entries **<** i.second) {  entries **=** i.second;  word **=** i.first;  }  }  cout **<<** entries **<<** endl;  cout **<<** word **<<** endl;  } |

Множества

Множество (set) - структура данных, предназначенная для хранения значений и быстрого выполнения на них базовых операций, включая поиск и удаление.

Свойства множеств, которые важны для решения олимпиадных задач:

* Элементы во множестве хранятся в отсортированном по возрастанию виде.
* Множество хранит только одно вхождение любого элемента (в отличие от мультимножества).

Большинство операций с множествами с С++ имеют сложность порядка O(logN)O(log⁡N) операций, однако также не стоит забывать про константу, которая эквивалентна константе map.

Один из способов интерпретации множеств заключается в том, что множества - это частный случай ассоциативных массивов, где значение отсутсвует, или не берётся во внимание. Это выражается в сходствах как в видах поддерживаемых операций, так и в способе реализации этих структур данных в стандартной библиотеке (красно-чёрные деревья).

Основные операции и асимптотическая сложность

* Вставка элемента – O(logN)O(log⁡N).
* Поиск элемента – O(logN)O(log⁡N).
* Удаление элемента – O(logN)O(log⁡N).
* Подсчёт элементов – O(1)O(1).

Использование

Решим простую задачу, используя множество:

Дан массив aa из n≤105n≤105 чисел ai≤109ai≤109. Необходимо вывести количество различных элементов в данном массиве.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **#include <bits/stdc++.h>**  **using** **namespace** std;  **int** **main**() {  *// Объявление множества:*  *// set<type>, где type - тип элементов.*  set**<int>** s;  **int** n, x;  cin **>>** n;  **for** (**int** i **=** 0; i **<** n; i**++**) {  cin **>>** x;  *// Вставка элементов массива во множество.*  s.insert(x);  }  *// Вывод размера множества,*  *// т.е. количества различных элементов в нем.*  cout **<<** s.size();  } |

Мультимножества

Мультимножество (multiset) обладает теми же свойствами, что и обычное множество, за исключением того, что один элемент может входить в мультимножество несколько раз. На мультимножества можно смотреть как на ассоциативные массивы, где значения – числа, обозначающие количество вхождений соответствующего ключа в мультимножество.

Структуры на хэш-таблицах

Для всех описанных выше структур данных существуют их аналоги с префиксом unordered\_ (неупорядоченный). Отличие их состоит в том, что они реализованы с использованием хэш-таблиц, а значит асимптотическая сложность операций на них равна O(1)O(1) (амортизированно), однако константа еще больше, чем в упорядоченных структурах, обсуждаемых выше.

Под “амортизированной” сложностью понимается, простыми словами, средняя сложность за большое количество операций. То есть даже при амортизированной сложности O(1)O(1), в одиночном худшем случае сложность может достигать O(n)O(n), однако происходит такое крайне редко.

Неупорядоченность проявляется в том, например, что элементы unordered\_map при итерировании не посещаются в порядке возрастания ключей. Это стоит учитывать при решении задач, где лишний логарифм в асимптотике может быть критичен.

Подключение библиотек

Если вы по какой-либо причине не используете bits/stdc++.h, то все упомянутые выше структуры данных находятся в одноименных библиотеках. Т.е. для использования map необходимо написать #include <map>.

**C++. map. Знакомство с синтаксисом**

* **map** — это такая структура данных, которая содержит пару **ключ** — **значение**

Ключом, как правило, является какое-то название, а значением назначаемые данные, связанные с этим названием. Например, у нас может быть набор объектов одной группы: яблоко, аплеьсин, персик", так вот название этой группы объектов — это ключ, а каждый из этих объектов — это то, что может быть значением, соотносящимся с названием группы. Это называется ассоциацией.

* **map** — это массив ассоциаций.

Чтобы объявить какой-то объект **map**, нужно определиться, какой тип будет иметь ключ, а какой тип будет у названий. А чтобы использовать **map**, следует включать соответствующую директиву:

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | //C++ Листинг #1 map  #include <iostream>  #include <map>    using namespace std;    int main(){      setlocale(LC\_ALL, "");      map<string, int> m;    //для ключей предопределяем тип string, а для данных int  } |

Если вы после всех объяснений плохо себе представляете, что такое **map**, но хорошо понимаете, что такое одномерный массив, попробуйте представить одномерный массив, у каждой ячейки которого появилось уникальное название. А использованием названия можно вытащить значение названной ячейки.

\*\*\*\*\*\*

**ФРУКТ**

**ЦВЕТ**

**СТРАНА**

МАССИВ

ЯБЛОКО

ЗЕЛЕНЫЙ

УКРАИНА

\*\*\*\*\*\*

**'Ауди'**

**'Фольцваген'**

**'Мерседес'**

МАССИВ

1000

900

3000

В один момент времени у ключа **map** может быть только одно значение (но этим значением может быть, например, вектор). Смысл в том, что это значение можно менять. Так, таблица с названиями автомобилей — это что-то вроде прайс-листа. Ключом являются названия фирм, а значенями числа (якобы цены). Цены можно обновлять, но нельзя задать несколько цен одному ключу. Давайте кодом отобразим одну из показанных таблиц:

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | //C++ Листинг #2 Знакомство с map  #include <iostream>  #include <map> //Будем использовать map    using namespace std;    int main(){      setlocale(LC\_ALL, "");      map<string, int> m;    //для ключей предопределяем тип string, а для данных int        m ={        {"Ауди", 1000},        {"Фольцваген", 900},        {"Мерседес", 3000},      };        //Теперь можно вытащить значение по ключу:       cout << m["Ауди"] << '\n';       cout << m["Фольцваген"] << '\n';       cout << m["Мерседес"] << '\n';  } |

Хоть **map** и походит на массив с именованными ячейками, это не масссив, это другая структура данных. Нельзя у **map** вытащить значения не по имени ячейки, а по индексу. Дело в том, что значения не размещаются по собственному порядку, поэтому у них индексов как таковых нет вообще. Значения упорядочиваются по ключам.

В **C++** у **map** ключ называется **first**, а назначаемое (назначенное) данное называется **second**. Обойти **map** можно почти как и обычный массив. В том смысле, что можно взять начало и конец объекта **map** и пройтись по этому диапазону, собирая все значения. Но в отличие от обхода массива (в котором ориентир идёт на номер хранимого элемента), обход **map** производится по позиции (с помощью перемещаемого по адресам итератора): итератор гуляет по ключам. Да и обращаться нужно, если к содержимому, то либо через итератор к ключу, либо через итератор к значению. Итератор — это такой бегунок, который позиционируется на адресе одного из элементов.

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | //C++ Листинг #3.1 Обход map  #include <iostream>  #include <map>    using namespace std;    int main(){      setlocale(LC\_ALL, "");      map<string, int> m;    //для ключей предопределяем тип string, а для данных int        m ={        {"Ауди", 1000},        {"Фольцваген", 900},        {"Мерседес", 3000},      };        map<string, int>::iterator it;      //объявили итератор      it = m.begin();           //установили его на начало map        while (it != m.end()){        cout << it -> first << ":   ";     //вытаскиваем ключ        cout << it -> second;              //вытаскиваем значение        cout << '\n';        it++;  //устанавливаем итератор на следующий элемент обходимой структуры данных      }  } |

Для наглядности цикл в листинге #3.1 был упрощён. Если не брать во внимание существование автоматического выведения типа, то вы можете использовать или встречать что-то вроде такого:

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //C++ Листинг #3.2 Обход map  map<string, int> m;    //для ключей предопределяем тип string, а для данных int        m ={        {"Ауди", 1000},        {"Фольцваген", 900},        {"Мерседес", 3000},      };       for (map<string, int>::iterator it=m.begin(); it!=m.end(); it++){        cout << it -> first << ":   ";     //вытаскиваем ключ        cout << it -> second;              //вытаскиваем значение        cout << '\n';     } |

Обычно ключ не предполагает редактирования собственного названия, поэтому напрямую до некоторых пор кодом ключу можно было задать название, использовать которое пермаментно, а если нужно было переименовать ключ, то находили нужную ячейку по ключу, сохраняли её значение в создаваемую, в создаваемой называли ключ нужным себе именем, после чего найденную ячейку удаляли, а вместо неё использовали созданную со скопированным значением. Меняются обычно значения, а не ключи. Вот несколько примеров изменить значения:

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | //C++ Листинг #4 map Меняем значения  #include <iostream>  #include <map>    using namespace std;    int main(){      setlocale(LC\_ALL, "");      map<string, int> m;    //для ключей предопределяем тип string, а для данных int        m ={        {"Ауди", 1000},        {"Фольцваген", 900},        {"Мерседес", 3000},      };       m["Ауди"] = m["Ауди"] + 77;   //Возросло на 77     m["Фольцваген"] = 1000;       //Обновилаось, стало 1000     m.find("Мерседес") -> second = 4000;  //и вот так можно (second <—-- значение)         for (auto it: m){        cout << it.first << ":  ";        cout << it.second << '\n';     }  } |

Основная сложность у меня во время моего знакомства была во вставке в **map** значений. Возможно, кто-то это читает только потому что испытывает те же трудности, что когда-то испытывал я. Дело в том, что для вставки **map** используют **pair**.

* **pair** — это шаблонный класс, который предоставляет возможность использовать объекты с двумя значениями.
* Для использования **pair** — нужно включать директиву **#include <utility>**.
* Отличаются **pair** и **map** тем, что один **pair** предоставляет один объект с двумя значениями (с одним ключ-значением), а один **map** множество пар ключ-значение.

Шаблонные классы — это такие классы, тип для объектов которых даётся только при объявлении объектов. **map** — это тоже шаблонный класс: когда мы объявляли объект **m** в наших листингах, мы сразу предопределяли ему, какие типы в нём будут. Для **pair** тоже нужно предопределять типы. В нашем случае в **map** вставляются **pair**, поэтому предопределённые типы **map** и **pair** должны быть идентичными.

* Для того, чтобы наделить объект пары названием и значением, следут использовать функцию **make\_pair**

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | //C++ Листинг #a1 Создание pair  #include <iostream>  #include <utility>    using namespace std;    int main(){      setlocale(LC\_ALL, "");        pair<string, int> p1;      pair<string, int> p2;      pair<string, int> p3;        p1 = make\_pair("Ауди", 1000);      p2 = make\_pair("Фольцваген", 1000);      p3 = make\_pair("Мерседес", 1000);  } |

Такие пары и вставляются в **map**:

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | //C++ Листинг #5.1 Вставляем (pair) в (map)  #include <iostream>  #include <utility>  #include <map>    using namespace std;    int main(){      setlocale(LC\_ALL, "");        pair<string, int> p1;      pair<string, int> p2;      pair<string, int> p3;        p1 = make\_pair("Ауди", 1000);      p2 = make\_pair("Фольцваген", 2000);      p3 = make\_pair("Мерседес", 3000);        map<string, int> m;    //для ключей предопределяем тип string, а для данных int      m.insert(p1);      m.insert(p2);      m.insert(p3);        for (auto it: m){        cout << it.first << ":  ";        cout << it.second << '\n';     }  } |

Если в **map** уже есть ключ с таким же именем, как у ключа вставляемой пары, то вставляемая пара будет проигнорирована. Нужно понимать, что ключи уникальны.

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | //C++ Листинг #5.2 Вставляем (pair) в (map)  #include <iostream>  #include <utility>  #include <map>    using namespace std;    int main(){      setlocale(LC\_ALL, "");        map<string, int> m;    //для ключей предопределяем тип string, а для данных int      m.insert(make\_pair<string, int>("Ауди", 1000));      m.insert(make\_pair<string, int>("МЕРСЕДЕС", 2000));      m.insert(make\_pair<string, int>("МЕРСЕДЕС", 5000));  // ключ "Мерседес" уже есть          for (auto it: m){               //Получилось только две ячейки        cout << it.first << ":  ";        cout << it.second << '\n';     }  } |

У ключа **map**, как и у значения, может быть любой тип. Но у типа должна обязательно быть определена операция <. Вот пример использования для ключа класса.

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | //C++ Листинг #6 Использование своего типа для ключа map  #include <iostream>  #include <map>  #include <utility>    using namespace std;    struct A{         //Структура для ключей     const string caption;     A(string caption): caption(caption){}  };    struct B{         //Cтруктура для значений      int value;  };    bool operator<(const A a,const A b){      return a.caption < b.caption;      //сравнение ключей для сортировки  }    int main()  {      map<A,B> m;        A a1("Ауди");      A a2("Мерседес");      A a3("Фольцваген");        B b1 = {1};      B b2 = {2};      B b3 = {3};        m.insert(make\_pair(a1, b1));      m.insert(make\_pair(a2, b2));      m.insert(make\_pair(a3, b3));        cout << "a1: " << m[a1].value << '\n';      cout << "a2: " << m[a2].value << '\n';      cout << "a3: " << m[a3].value << '\n';      cout << "======================\n";        for (const auto &i:m){        cout << i.second.value << '\n';      }  } |

Операрация < используется для сортировки ключей, поэтому она должна быть определена так, чтобы в результате сравнения разные ключи не воспринимались одинаково, иначе, если разные ключи воспринимаются одинаковыми ключами, вставка в **map** просто дальше первой происходить не будет.